Recap:

Recurrence Relations

- Recursion Tree (informal but helpful)
- Substitution Method (formal, need to guess asymptotics first)
- Master Theorem (for recurrence of the following form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n); \quad a \geq 1, \ b > 1$$)

1. Master Theorem (cont'd)

Master Theorem:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n); \quad a \geq 1 \quad b > 1$$

Case 1: $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$

$$\Rightarrow T(n) = \Theta(n^{\log_b a})$$

Case 2: $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ $\quad k \geq 0$

$$\Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

Case 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$

and $af(n/b) \leq c \cdot f(n)$ for some $c < 1$ (regularity condition)

$$\Rightarrow T(n) = \Theta(f(n))$$

Intuition: let $Q = n^{\log_b a}$, then compare it with f:

Case 1: f polynomially smaller than Q

Case 2: f is larger than Q by a polylog factor

Case 3: f is polynomially larger than Q + regularity

Examples:

① $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

$Q = n^{\log_b a} = n^{\log_2 2} = n$

$\Theta(n) / n = \Theta(1) = \Theta(\log^0 n)$

Case 2! $\Rightarrow T(n) = \Theta(n \log n)$

② $T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$

$Q = n^{\log_b a} = n^{\log_2 7} \approx n^{2.8}$

$\frac{\Theta(n^2)}{n^{2.8}} = \Theta(n^{-0.8}) = O(n^{-0.8})$

Case 1! $\Rightarrow T(n) = \Theta(n^{\log_2 7})$

③ $T(n) = 4T(n/2) + n^3$

$Q = n^{\log_b a} = n^{\log_2 4} = n^2$

$\frac{n^3}{n^2} = n = \Omega(n)$ Case 3?

$af(\frac{n}{b}) = 4 \cdot (\frac{n}{2})^3 = \frac{n^3}{2} \leq \underset{\substack{\uparrow \\ c<1}}{(\frac{3}{4})} \cdot n^3$

Case 3! $\Rightarrow T(n) = \Theta(n^3)$

2. Quick-Sort

Intuition:

Divide
1. Select a pivot (many ways of doing this)
2. Partition: smaller than pivot to the left, others to the right (of pivot)

Conquer
3. Recursively sort both halves

```
QuickSort(A[1,...,n])
    k = Partition(A) // k is index of pivot
    QuickSort(A[1,...,k-1])
    QuickSort(A[k+1,...,n])
```

```
Partition(A[1,...,n])
    pivot = A[n]
    i = 1
    For j = 1 to n-1
        If A[j] ≤ pivot
            Swap A[i] with A[j]
            i = i + 1
        EndIf
    EndFor
    Swap A[i] with A[n]
    Return i
```

e.g.

pivot = 4

i :

j :

5	1	6	3	2	4

$\Downarrow$

5	1	6	3	2	4

$\Downarrow$

1	5	6	3	2	4

$\Downarrow$

1	5	6	3	2	4

$\Downarrow$

1	3	6	5	2	4

$\Downarrow$

1	3	2	5	6	4

$\Rightarrow$

1	3	2	4	6	5

3. Analysis of Quick-Sort

Correctness of Partition

Proof:

Loop Invariant

1. For all $k \in \{1, \dots, i-1\}$, $A[k] \le pivot$
2. For all $k \in \{i, \dots, j-1\}$, $A[k] > pivot$

$$\underbrace{1\ 2\ \dots\ i-1}_{\le pivot}\ \underbrace{i\ \ i+1\ \dots\ j-1}_{> pivot}\ \underbrace{j\ \ j+1\ \dots\ n}_{??}$$

Initialization: $i = j = 1$, trivially true ✓

Maintenance: Assume $A[1, \dots, i-1] \le pivot$
and $A[i, \dots, j-1] > pivot$

$$\begin{array}{|c|c|c|} \hline 1 \quad \dots \quad i-1 & i \quad \dots \quad j-1 & j \\ \hline \multicolumn{1}{c}{\le pivot} & \multicolumn{1}{c}{> pivot} & \multicolumn{1}{c}{?} \end{array}$$

① if $A[j] > pivot$,

$$\begin{array}{|c|c|c|} \hline 1 \quad \dots \quad i-1 & i \quad \dots \quad j-1 & j \\ \hline \multicolumn{1}{c}{\le pivot} & \multicolumn{2}{c}{> pivot} \end{array}$$

② if $A[j] \le pivot$,
the new $A[j]$ is the old $A[i] > pivot$
the new $A[i]$ is the old $A[j] \le pivot$

$$\underbrace{\boxed{1 \quad \dots \quad i-1} \boxed{i}}_{\le pivot} \underbrace{\boxed{i+1 \;\cdots\; j-1} \boxed{j}}_{> pivot}$$

Termination: Right after For loop, we have $j = n$, so:
① $A[1, \dots, i-1] \le pivot$,
② $A[i, \dots, n-1] > pivot$,
③ $A[n] = pivot$
After line 9 (Swap $A[n]$ with $A[i]$):
① $A[1, \dots, i-1] \le pivot$,
② $A[i] = pivot$
③ $A[i+1, \dots, n] > pivot$ #

Runtime Analysis of Partition: $\Theta(n)$

Correctness of Quick-Sort: follows correctness of Partition

Runtime Analysis of Quick-Sort:

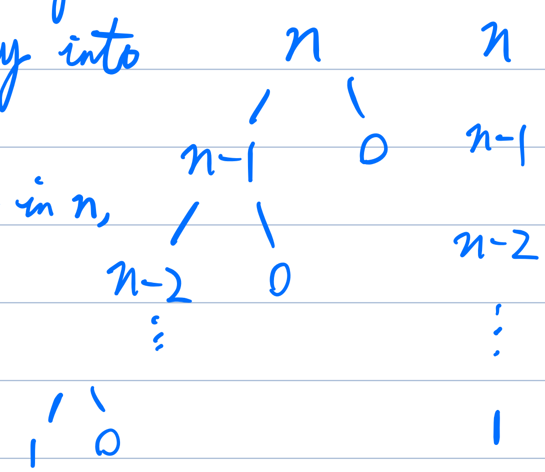
Worst Case:

Happens when array already sorted (!!!)

Each Partition splits the array into $(n-1)$ and 0.

Since Partition takes time linear in n, overall runtime is

$n + (n-1) + \cdots + 1 = \Theta(n^2)$.

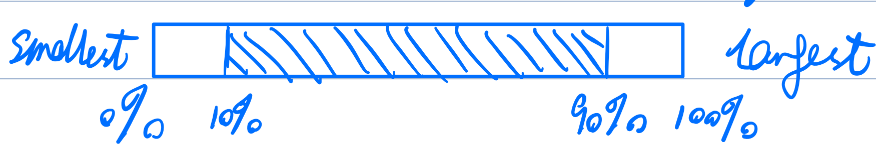


Best Case: Every time split exactly in half.

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Same as Merge Sort: $\Theta(n \log n)$

Average Case (roughly): if the array we receive is randomly shuffled, we can expect the last element (pivot) to be not at the top or bottom percentiles, e.g. there is 80% chance that it is between 10% – 90%. (80% prob. to be in ▨)



Assume for simplicity we always get a 90%/10% split:

$$T(n) = T\left(\tfrac{1}{10}n\right) + T\left(\tfrac{9}{10}n\right) + \Theta(n)$$

$$\log_{\frac{10}{9}} n \left\{ \begin{array}{lr} n & n \\ \frac{9}{10}n \quad \frac{1}{10}n & n \\ \frac{81}{100}n \quad \frac{9}{100}n \quad \frac{9}{100}n \quad \frac{1}{100}n & n \\ \cdots\cdots \qquad \cdots\cdots & \le n \\ 1 \quad 1 \quad 1 \quad \cdots\cdots & \le n \end{array} \right.$$

$$1 = \left(\tfrac{9}{10}\right)^k \cdot n \Rightarrow k = \log_{\frac{10}{9}} n$$

Runtime is: $n \cdot \log_{\frac{10}{9}} n$

$$= n \cdot \frac{\log n}{\log \frac{10}{9}} \approx 6.6 \cdot n \log n = \Theta(n \log n)$$

Quick-Sort performs poorly on some inputs, One way to overcome this is by picking the pivot randomly. This way, w/ prob. 80% we get a partition of at least (10%, 90%)-balanced. This randomized Quick-Sort algorithm has expected runtime of $\Theta(n \log n)$ for all inputs (i.e. in worst case)

In practice, we can pick the elem. in the middle, or the median of {first, middle, last}, or just the overall median.

Quick-Sort:

- $\Theta(n^2)$ worst case (worse than MergeSort!)
  $\Theta(n \log n)$ best/average case
- Unstable
- In place
- The constant in $\Theta(\cdot)$ is quite low, so works very well in practice!
- Based on Divide & Conquer
- Sort of "MergeSort in Reverse"